MORALE METHODOLOGY GUIDEBOOK
Functional Modeling and Model-Based Architectural Redesign of Software Agents

Ashok K. Goel
J. William Murdock
College of Computing
Georgia Institute of Technology
{goel,murdock}@cc.gatech.edu

6. Introduction

Building a computational theory of evolutionary software design involves addressing three questions pertaining to adaptation: *why*, *what* and *how*. The first question asks why might the design of a software artifact be adapted? Potential answers include failure of the artifact to meet its task requirements and evolution of its task requirements. The second question asks what in the artifact might be adapted? Potential answers include modifications to the artifact's information-processing architecture and to its domain knowledge. The third question asks how might the needed modifications be identified? Potential answers include use of models of the artifact's architecture and knowledge, specifications of its design rationale, and descriptions of its processing in specific scenarios. In this article, we are interested in these three issues for a class of software artifacts characterized by sequential and centralized processing. That is, the class of software artifacts of interest here does not cover parallel processing or distributed systems. The goal is to develop a computing environment for supporting (human) software engineers in interactive redesign of this class of software artifacts.

Building a computing environment for interactive evolutionary design of software artifacts raises an additional issue pertaining to autonomy. At one extreme, the software engineer may make all redesign decisions and realize the modified design by reprogramming the artifact. At this end, the computing environment may provide access to relevant design knowledge, e.g., the design rationale, software architecture, and program code, but leave all decision making to the software engineer. At the other end, the artifact may be self-adaptive, autonomously redesigning its own architecture and knowledge, and reprogramming its own code. At this extreme, the software engineer may monitor the processes of self-redesign and self-programming, and intervene if and when needed. In this article, we are interested in the middle of this autonomy spectrum, where the artifact makes the redesign decisions but leaves the reprogramming to the software engineer. Thus the artifact is an *intelligent software agent* that is capable of making decisions about its own redesign.

We describe a methodology called MESA (for Model-based Evolution in Software Agents) that seeks to address the above class of problems in evolutionary software design. The MESA methodology has two parts: (1) modeling of the information-processing architecture and domain knowledge of a software agent, and (2) use of the model for architectural and knowledge redesign. In this article, we focus on the modeling of the architecture and knowledge of the agent, and the use of the model for architectural redesign. We describe a modeling method called SBF (for Structure-Behavior-Function) that views a software agent as a telelogical artifact, i.e., as an artifact that has particular functions and specific behaviors that arise from the structure of the artifact and result in the accomplishment of its functions. The SBF modeling method leads to a Task-Method-Knowledge (or TMK) model of the software agent, which specifies both the functional architecture of agent in terms of its task-method structure, and its domain knowledge. We also outline the use of the TMK model for an example pertaining to MOSAIC.

7. Structure-Behavior-Function Modeling

The SBF method of modeling arises from our earlier work on conceptual phase of functional design of physical devices [Goel 1991a, 1991b, Goel and Chandrasekaran 1989, 1992]. The SBF model of a device specifies how the device works, i.e., how its structure delivers its functions. In particular, the SBF model explicitly represents not only the structure and functions of the device, but also its internal behaviors that explain how the functions of the structural components of the system get composed into the functions of configurations of components, and how the functions of the component configurations get recursively composed into the functions of the device as a whole. Thus the SBF model is organized in a hierarchy of the form *function -> behavior -> function -> behavior* ..., up to any desired level of component aggregation and abstraction. The functions at any level in this hierarchy act as indices into behaviors that achieve them, and the behaviors act as indices into functions of component configurations at the next lower level.

The SBF language provides a general vocabulary for representing the structure, behavior and functions of physical devices. The representation of a structure specifies its topology in terms of components, substances that can flow among the components, and relations among the components and substances. The representation of a function of a structure specifies the behavioral state the structure takes as input and the behavioral state it gives as output, where a state is specified in terms of properties of components and substances. The representation of a behavior specifies a sequence of behavioral states and state transitions that results in the function of the structure, where the transitions are specified in terms of functional, causal, and structural relations of the substructures.

In case of intelligent software agents, the SBF modeling method results in a TMK model of the agent's functional architecture and domain knowledge. Figure 1 illustrates the organization of the TMK model of the portion of Mosaic responsible for external viewing of Postscript files. The model describes the architecture of the system in terms of its task-method structure and domain knowledge. Tasks (e.g., Display-Interpreted-File) constitute the building blocks of the system's architecture. Methods (e.g., External-Display) decompose a task into subtasks, which, in turn, are recursively decomposed into smaller and simpler subtasks. The leaf tasks (e.g., Access-Display-Program) are directly accomplished by "procedures" that use domain knowledge and get encoded in the program. A task is characterized as a transformation from an input information-state to an output information-state. It is specified by the type(s) of information it consumes as input and produces as output, and specific relations that hold true among the task's inputs and outputs. In Mosaic, a task such as Display-Interpreted-File actually is a family of tasks, whose members differ from each other only in specifics of the information they take as input and give as output. but not in the relations between the input and output information. A method is characterized as a (partially ordered) sequence of information-states and information-state transitions that compose tasks at one level into a task at the next higher level in the task structure. It specifies the data and control dependencies between the tasks at the lower level. Again, in Mosaic, a method such as External-Display really is a family of methods, whose members differ from each other only in specifics of the information in the state transitions, but not in the data/control dependencies among the lower-level tasks. As Figure 1 indicates, S(tructure) in SBF corresponds to the leaf tasks, the B(ehavior) to the methods, and the F(unction) to the tasks in the task structure.

8. The Task-Method-Knowledge Language

Autognostic [Stroulia and Goel 1994a, 1994b, 1995, 1996] is a semi-formal language for specifying TMK models of intelligent software agents. Its TMK language enables a specification of the tasks and subtasks of the agent, the methods used by the agent to accomplish a task, and the domain knowledge used by the methods. The tasks specify the input and output information states. The methods specify the subtasks of a task and the flow of data and control between the subtasks. The domain knowledge is specified in terms of domain concepts, relations and constraints. Thus the TMK language provides primitives for specifying the functional semantics of tasks (including the leaf tasks), the compositional semantics of methods, and also the "causal" (data/control) semantics of the interdependencies among tasks, methods, and domain knowledge. The following specifications of the TMK language are taken due to Stroulia [1994].

8.1 The Task

Each task is described as a tuple $T := (in, out, proto, by, proc, cond, sem)$, [Footnote: In addition to these slots, there are two additional slots which we have considered *subtask_of* and *instances* which are important to building a trace of a model but are not exactly parts of the model itself.] where *in* is the input of the task, *out* is its output, *proto* is another task of which the current task is an instance, *by* is a set of methods which can be applied to accomplish the task, *proc* is a procedure [Footnote: The term *procedure* here suggests that they are performing an internal manipulation of knowledge. Alternatively, the can be viewed as *operators*, i.e. as primitive actions which have a direct impact on some external world state.] which can be applied to accomplish the task, *cond* is a set of conditions under which the task should be performed, and *sem* is a set of semantic relations partially describing the role of the task in the task structure.

The slots *in* and *out* are information states. For some specific task tk, the information state $in(tk)$ consists of the types of information which are necessary for the task tk to be accomplished. Thus, for each type of information, $i \in in(tk)$, i is either required by some of the subtasks resulting from the decomposition of tk, by some method $m \in by(tk)$, or it is an input parameter of the procedure which implements $proc(tk)$. For the same task tk, the information state $out(tk)$ consists of the types of information which are produced by the task tk, and which are going to be used by other subsequent tasks.

The slots $cond(tk)$ and $sem(tk)$ are sets of relations. The relations in $cond(tk)$ describe the conditions under which the task tk contributes to the progress of reasoning. They are used as a criterion to decide whether to perform the task or not; thus they refer to the types of information available in $in(tk)$. The relations in $sem(tk)$ partially describe the information transformation that tk performs in the service of the overall reasoning. Essentially, the describe the function intended of the task in the overall context of reasoning. Thus each relation $sem(tk)$ is either a unary relation on some type of information in $out(tk)$ or it relates to some type of information in $out(tk)$ with some type of information in $in(tk)$.

The $by(tk)$ slot is a set of methods which are applicable to the task tk. When one of these methods is applied to tk, it sets up a set of subtasks such that if each subtask is accomplished, the overall information transformation intended by tk will be accomplished. The attribute *by* provides a partonomic hierarchy among tasks.

Finally, the $proto(tk)$ slot is another task, tp, of which tk is an instance. It is simply a task which accomplishes an information transformation more general than or equivalent to the transformation accomplished by tk. If the task tk does not have associated with it a method or a procedure, i.e. $by(tk) = \emptyset \wedge proc(tk) = \emptyset$, then it can be accomplished with the methods or the procedures associated with $tp = proto(tk)$. The attribute *proto* provides a taxonomic hierarchy among tasks.

In Autognostic the SBF language is implemented in a schema-based framework. Table 1 depicts the schema for specifying a task.

8.2 The Method

Each method is described as a tuple $M := (app,cond,sub,ctrl)$, where *app* is the task to which M is applied, i.e., the task which M is intended to accomplish, *cond* is a set of conditions under which the method is applicable, *sub* is the set of subtasks that the method sets up for *app*, and *ctrl* is a set of control relations over these subtasks.

For any specific method, mk, the $cond(mk)$ slot is a set of relations which describe the conditions under which the task mk contributes to the progress of the reasoning. They are used as a criterion to decide whether to perform the task or not; thus they refer to the types of information available in the information state $in(tk)$.

When a method, mk, is applied to the task $app(mk)$ it decomposes it into a set of simpler subtasks $sub(mk)$. The order in which these subtasks must be accomplished is not completely specified by the method. Instead, the method, mk imposes only a partial ordering among them which is specified in the *ctrl* slot of the method schema. In the SBF language, there are three control operators *serially*, *in_parallel*, and *loop_until*. The control that the method imposes over the subtasks it sets up, $ctrl(mk)$, is specified as a control operator imposed over a set of subtasks in the set $sub(mk)$ or other control statements. When two subtasks are connected by the *serially* operator, they must be performed in the order they are mentioned. When they are connected by the *in_parallel* operator, they may be performed in any ordering. Finally, when they are connected by the *loop_until* operator, then they must be performed repeatedly until the condition annotating the *loop_until* operator is met.

Table 2 depicts the schema for specifying a method,

8.3 The Type of Information

The input and output of each task in the problem-solver's task structure consists of a set of types of information. A type of information is defined as a tuple $I := (obj,in,out)$, where *obj* is the type of domain concept of which this information type is an instance, *in* is the set of tasks which take the type of information as an input, *out* is the set of tasks which produce the type of information as an output. Note that for a given information type, i, it is required that $\forall\ tk \in in(i).\ i \in in(tk)$ and $\forall\ tk \in out(i).\ i \in out(tk)$.

Table 3 depicts the schema for specifying a type of information reasoned about by the problem-solver's task structure.

8.4 The Domain Concept

Domain concepts are the concepts which are relevant to reasoning in a particular domain. They may correspond to physical entries in the real world or they may be abstract. A domain concept is described as a tuple $DConc := (dom,attr,id)$ where *dom* is the domain in which this concept may take values, *attr* is the set of attributes characteristic of the concept, and *id* is a predicate that evaluates the identity between two instances of a concept.

The actual knowledge of the problem-solver about domain objects of this concept type is in the form of a set of instances that the problem-solver knows about. This set of instances constitutes the domain, *dom* of this object type.

Additionally, for each attribute in *attr*, the SBF language specifies its *name*, its *type*, and a function f which given a specific instance of the concept produces the value of the attribute for the given instance.

Table 4 depicts the schema for specifying a type of domain concept.

8.5 The Domain Relation

A domain relation is specified by a tuple $DRel := (in,out,truth,pred,invpred,idx)$. The elements *in* and *out* are the types of arguments that the relation takes as input and produces as output. The elements *truth*, *pred*, and *invpred* refer to the actual knowledge of the problem-solver regarding this domain relation. For a given domain relation rk, this knowledge may have the form of a truth table, $truth(rk)$ which includes all of the pairs $(ij,oj) \in in(rk) \times out(rk)$ for which the relation is true. Alternative, there may be a predicate, *pred*(*rk*), which, when applied to any instance $ij \in in(rk)$, returns the one unique $oj \in out(rk)$ for which the relation is true and *nil* if none exists. In the latter case, the problem solver may also know of an inverse predicate *invpred*(*rk*) which when applied to any instance $oj \in out(rk)$, returns the one unique $ij \in in(rk)$ for which the relation is true and *nil* otherwise. A key difference between these two ways of evaluating *rk* is that in the first case the contents of *rk* are inspectable; the problem-solver may modify *rk* by simply adding or deleting tuples in *truth*(*rk*). In the latter case, since the predicates *pred*(*rk*) and *invpred*(*rk*) are non-inspectable "definitions", the problem solver cannot modify them.

Finally, the attribute *idx*(*rk*) characterizes the type of domain relation. The concepts in the domain of the problem-solver may be related through domain relations. Domain relations can be of two types: relations whose truth value depends on the state of the world, which we call *state-of-the-world* relations, and organizational relations which are "invented" to increase the efficiency of problem solving, which we call *convention* relations. A problem-solver may often create associations between different types of knowledge, which are all relevant at the same point in its problem solving, in order to be able to retrieve the appropriate pieces of information more easily. An important difference between these two kinds of relations is that the evidence necessary for the problem-solver to modify the convention relations is less than the evidence necessary to modify the state-of-the-world relations. In fact, the problem-solver may begin without any specific organization of its knowledge, i.e. with an undefined domain relation, and it can induce this relation by adding entries into its table on an as needed basis, i.e. every time it fails because of its incomplete organization.

Table 5 depicts the schema for specifying a domain relation.

8.6 The Domain Constraint

Constraints are higher-order relations that apply to the objects of a domain. A domain constraint is specified as a tuple $DCons := (if, then, where)$ where *if* and *then* are domain relations and *where* is a predicate which connects the attributes of those two. If in some particular domain, there is a constraint $(r1(A,B), r2(B,A))$ where $r1$ and $r2$ are relations in that domain and A and B are types of objects in that domain, then this constraint means that if for a particular tuple (a,b) the relation $r1$ is true, then the relation $r2$ is true for (b,a). If the types of attributes of $r1$ and $r2$ are not the same, it is still possible to specify a constraint between these two relations if there is a predicate *where* such that it applies tot he attributes of relation $r1$ and produces the attributes of $r2$ which do not belong in the set of attributes of $r1$.

Table 6 depicts the schema for specifying a domain constraint.

9. Model-Based Software Evolution

A common problem in evolutionary design of intelligent software agents is incremental addition of new functionalities desired of an agent. In this class of problems, a software agent delivers a set F(old) of functions f(1),...,f(n-1), but subsequently the software engineer (or a user) specifies a new set, F(new), containing an additional function, f(n). The specific problem of adding the functionality of supporting user-specified external viewers in Mosaic 2.4 is an instance of this class of problems. Often the additional function, f(n) in F(new), is similar and related to some function f(i) in F(old). For example, while Mosaic enables external viewing of Postscript files using Ghostview, a user may specify the additional functionality of viewing PDF files using the Acrobat Reader, where the new functionality is similar to the one already delivered. The design issue in this class of problems becomes the adaptation of the current structure of the software system, S(old), that delivered F(old), into a modified structure, S(new), for delivering F(new). To help maintain the global consistency of the behaviors of the software system, and to make the implementation of modifications easy, the adaptation strategy needs to follow the principle of minimal change, insuring that the modifications are as localized as possible.

The TMK model of the software agent enables multiple strategies for modifying S(old) to obtain S(new). Here we outline one strategy that we will call *within-problem-analogy*: (1) The function f(i) in F(old) that most closely matches the additional function f(n) in F(new) is identified. In the example of viewing PDF files, this may result in the identification of viewing Postscript files as the closest match. The representation of the functions in the SBF language enables this matching. (2) The components and the behaviors responsible for achievement of f(i) are localized and identified. In our running example, the components of Access-Display-Program, Execute-Display-Program, and Library-of-Display-Programs are identified, along with the behavior of External-Display that composes the functions of these components into the function Display-Interpreted-File for Postscript files. The functional, compositional and causal semantics of the SBF model enable this localization and identification. (3) The identified components and behaviors are abstracted into a Generic Teleological Mechanism (GTM) [Stroulia and Goel 1992; Bhatta and Goel 1993] by removing all structural information specific to display of Postscript files. For example, in the GTM, the components Access-Display-Program and Execute-Display-Program are specified only by their functions, and the functional specifications refers only to types of information they take as input and give as output, not to the specifics of information relevant to Postscript files. A GTM thus specifies a functional, compositional and causal design pattern. The SBF language provides the vocabulary for representing GTMs. (4) The abstracted GTM is instantiated in the context of the additional function f(n). In our example, the GTM for external viewers is instantiated in the context of viewing PDF files. This instantiation results in the placement of a new display program (the Acrobat Reader, specified as part of f(n)) in the Library-of-Display-Programs, and the introduction of a new method for viewing PDF files in the family of External-Display methods. The SBF model focuses the GTM instantiation. In addition, the original SBF model is now revised to reflect the modified structure for the software system. We call this strategy within-problem analogy because in solving a subproblem (e.g., external viewing of PDF files), it relies on making an analogy to the solution to another subproblem (e.g., external viewing of Postscript files).

10. Relationship to MORALE

The MESA methodology for software evolution is related to the MORALE project in three ways:

1. Architectural Description Language:TMK is a language for describing the functional architecture and domain knowledge of a software artifact. The TMK model of a software artifact is organized in a hierarchy of functional and compositional abstractions, and explicitly relates the behaviors of the artifact with the domain knowledge.
2. Model-Based Architectural Redesign:MESA provides some answers to the *why*, *what* and *how* questions we raised in the introduction. In particular, the TMK model of a software artifact explicitly represents what can be potentially modified, which ranges from the tasks, methods, domain knowledge, and flow of data and control. It also enables "how-to" strategies, such as within-problem analogy, that identify the needed modifications to the architecture of the software artifact.
3. Functional Modeling Method:The SBF method for functional modeling provides a set of functional, behavioral and compositional abstractions of software artifacts. These abstractions will help guide the process of reverse engineering of legacy systems.

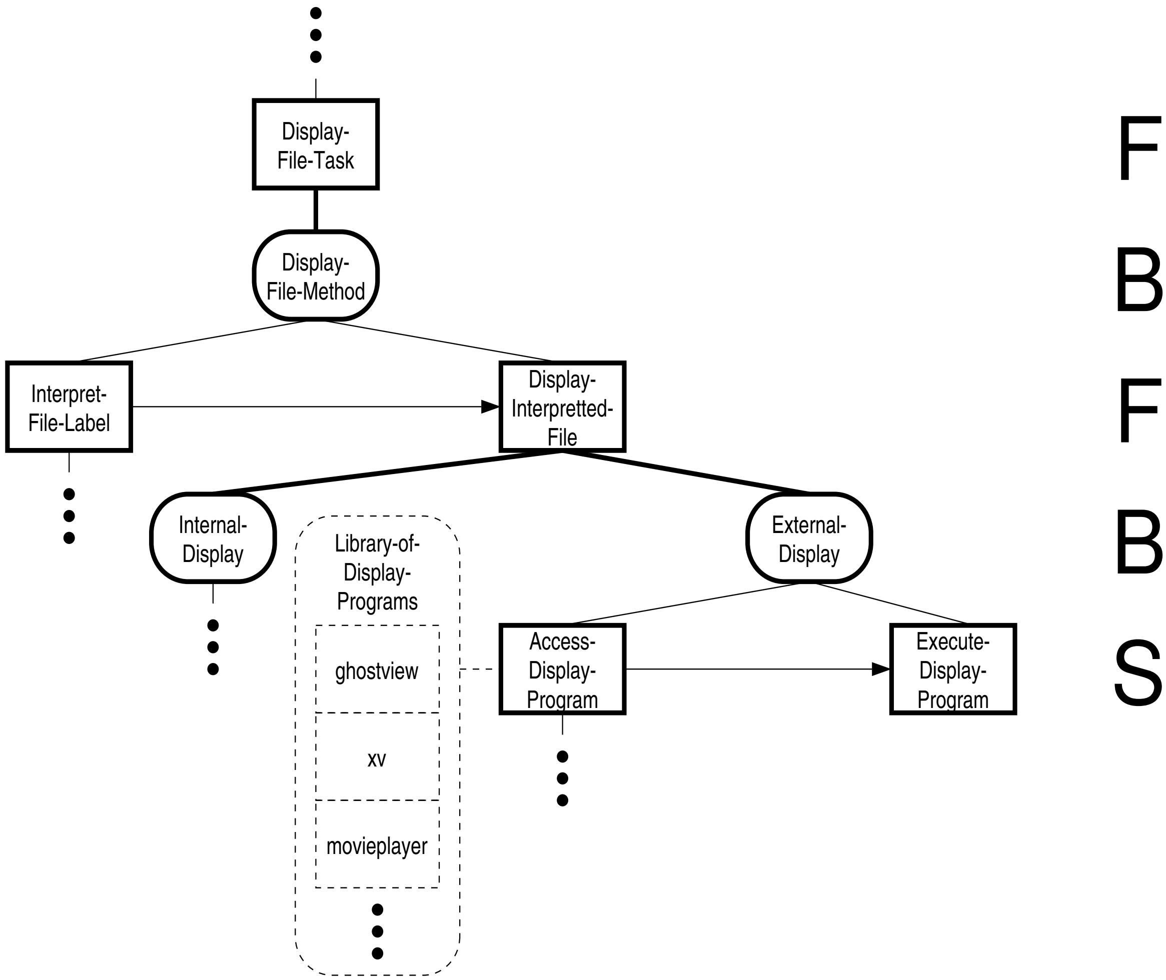

Figure 16 A partial SBF model of a Mosaic 2.4. Rectangular boxes indicate tasks while rounded boxes indicate methods. Knowledge sources are contained within dotted lines. Thick lines between boxes represent the availability of methods to accomplish a task; thin lines represent a method's composition of subtasks; Horizontal arrows indicate ordering of subtasks within a method.

task	the name of the task t
input	the types of information taken as input $in(t)$
output	the types of information produced as output $out(t)$
prototype	the prototype of which this is an instance $proto(t)$
by_methods	the set of methods possibly applicable $by(t)$
procedure	the procedure that directly accomplishes this $proc(t)$
under_condition	the conditions under which this is useful $cond(t)$
semantics	the information transformation performed $sem(t)$

Table 43 The Task Schema

method	the name of the method m
applied_to	the task for which this is applicable $spp(m)$
under_condition	the conditions under which this is useful $cond(m)$
subtasks	the set of subtasks that this sets up $sub(m)$
control	the flow of control among the subtasks $ctrl(m)$

Table 44 The Method Schema

information	the name of the information type i
input_to	the tasks that consume this as an input $in(i)$
produced_by	the tasks that produce this as an output $out(i)$
object_type	the type of domain concept of which this is an instance $obj(i)$

Table 45 The Type of Information Schema

concept	the name of the concept c
domain	the domain in which instances of this take values $dom(c)$
attributes	the set of attributes applicable to this $attr(c)$
identity_test	the predicate for identity between instances of this $id(c)$

Table 46 The Domain Concept Schema

relation	the name of the domain relation *r*
input_args	the types of concepts taken as input *in*(*r*)
output_args	the types of concepts produced as output *out*(*r*)
truth_table	the table enumerating the tuples are valid *truth*(*r*)
predicate	the predicate for deriving valid output from input *pred*(*r*)
inverse_predicate	the predicate for deriving valid input from output *pred*(*r*)
indexing_rel	whether or not this is used for indexing purposes only *idx*(*r*)

Table 47 The Domain Relation Schema

constraint	the name of the domain constraint *c*
if	one of the relations involved *if*(*c*)
if	the other relation involved *then*(*c*)
where	the predicate relating non-common attributes of the relations *where*(*c*)

Table 48 The Domain Constraint Schema